# Blind Backdoors in Deep Learning Models


Eugene Bagdasaryan
*Cornell Tech*
eugene@cs.cornell.edu

Vitaly Shmatikov
*Cornell Tech*
shmat@cs.cornell.edu

## Abstract

We investigate a new method for injecting backdoors into machine learning models, based on compromising the loss-value computation in the model-training code. We use it to demonstrate new classes of backdoors strictly more powerful than those in the prior literature: single-pixel and physical backdoors in ImageNet models, backdoors that switch the model to a covert, privacy-violating task, and backdoors that do not require inference-time input modifications.

Our attack is blind: the attacker cannot modify the training data, nor observe the execution of his code, nor access the resulting model. The attack code creates poisoned training inputs "on the fly," as the model is training, and uses multi-objective optimization to achieve high accuracy on both the main and backdoor tasks. We show how a blind attack can evade any known defense and propose new ones.

## 1 Introduction

A *backdoor* is a covert functionality in a machine learning model that causes it to produce incorrect outputs on inputs containing a certain "trigger" feature chosen by the attacker. Prior work demonstrated how backdoors can be introduced into a model by an attacker who poisons the training data with specially crafted inputs [5, 6, 28, 92], or else by an attacker who trains the model in outsourced-training and model-reuse scenarios [40, 55, 58, 98]. These backdoors are weaker versions of UAPs, universal adversarial perturbations [8, 61]. Just like UAPs, a backdoor transformation applied to any input causes the model to misclassify it to an attacker-chosen label, but whereas UAPs work against unmodified models, backdoors require the attacker to both change the model *and* change the input at inference time.

***Our contributions.*** We investigate a new vector for backdoor attacks: *code poisoning*. Machine learning pipelines include code from open-source and proprietary repositories, managed via build and integration tools. Code management platforms are known vectors for malicious code injection, enabling attackers to directly modify source and binary code [7, 19, 67].

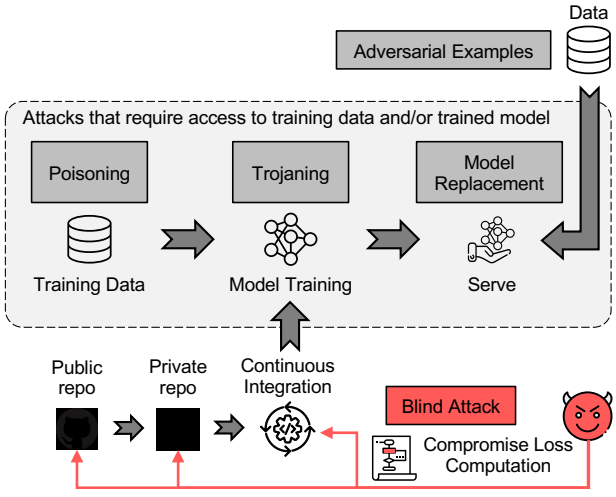



Figure 1: **Machine learning pipeline**.

Source-code backdoors of the type studied in this paper can be discovered by code inspection and analysis. Today, even popular ML repositories [33, 42, 62, 96], which have thousands of forks, are accompanied only by rudimentary tests (such as testing the shape of the output). We hope to motivate ML developers to carefully review the functionality added by every commit and design automated tests for the presence of backdoor code.

Code poisoning is a *blind* attack. When implementing the attack code, the attacker does not have access to the training data on which it will operate. He cannot observe the code during its execution, nor the resulting model, nor any other output of the training process (see Figure 1).

Our prototype attack code[1] synthesizes poisoning inputs "on the fly" when computing loss values during training. This is not enough, however. A blind attack cannot combine main-task, backdoor, and defense-evasion objectives into a single loss function as in [3, 84] because (a) the scaling coefficients are data- and model-dependent and cannot be precomputed

[1] Available at https://github.com/ebagdasa/backdoors101.

by a code-only attacker, and (b) a fixed combination is suboptimal when the losses represent different tasks.

We view backdoor injection as an instance of *multi-task learning for conflicting objectives*—namely, training the same model for high accuracy on the main and backdoor tasks simultaneously—and use Multiple Gradient Descent Algorithm with the Franke-Wolfe optimizer [16, 81] to find an optimal, self-balancing loss function that achieves high accuracy on both the main and backdoor tasks.

To illustrate the power of blind attacks, we use them to inject (1) single-pixel and physical backdoors in ImageNet; (2) backdoors that switch the model to an entirely different, privacy-violating functionality, e.g., cause a model that counts the number of faces in a photo to covertly recognize specific individuals; and (3) semantic backdoors that do not require the attacker to modify the input at inference time, e.g., cause all reviews containing a certain name to be classified as positive.

We analyze all previously proposed defenses against backdoors: discovering backdoors by input perturbation [95], detecting anomalies in model behavior on backdoor inputs [12], and suppressing the influence of outliers [32]. We show how a blind attacker can evade any of them by incorporating defense evasion into the loss computation.

Finally, we report the performance overhead of our attacks and discuss better defenses, including certified robustness [27, 71] and trusted computational graphs.

## 2 Backdoors in Deep Learning Models

### 2.1 Machine learning background

The goal of a machine learning algorithm is to compute a model $\theta$ that approximates some task $m : \mathcal{X} \to \mathcal{Y}$, which maps inputs from domain $\mathcal{X}$ to labels from domain $\mathcal{Y}$. In supervised learning, the algorithm iterates over a training dataset drawn from $\mathcal{X} \times \mathcal{Y}$. Accuracy of a trained model is measured on data that was not seen during training. We focus on neural networks [25]. For each tuple $(x, y)$ in the dataset, the algorithm computes the *loss value* $\ell = L(\theta(x), y)$ using some criterion $L$ (e.g., cross-entropy or mean square error), then updates the model with the gradients $g = \nabla\ell$ using backpropagation [74]. Table 1 shows our notation.

### 2.2 Backdoors

Prior work [28, 55] focused on universal pixel-pattern backdoors in image classification tasks. These backdoors involve a normal model $\theta$ and a backdoored model $\theta^*$ that performs the same task as $\theta$ on unmodified inputs, i.e., $\theta(x) = \theta^*(x) = y$. If at inference time a certain pixel pattern is added to the input, then $\theta^*$ assigns a fixed, incorrect label to it, i.e., $\theta^*(x^*) = y^*$, whereas $\theta(x^*) = \theta(x) = y$.

We take a broader view of backdoors as an instance of *multi-task learning* where the model is simultaneously trained for its original (main) task and a backdoor task injected by the attacker. Triggering the backdoor need not require the adversary to modify the input at inference time, and the backdoor need not be universal, i.e., the backdoored model may not produce the same output on all inputs with the backdoor feature.

Table 1: **Notation.**

| Term | Description |
|---|---|
| $\mathcal{X} \times \mathcal{Y}$ | domain space of inputs $\mathcal{X}$ and labels $\mathcal{Y}$ |
| $m : \mathcal{X} \to \mathcal{Y}$ | learning task |
| $\theta$ | normal model |
| $\theta^*$ | backdoored model |
| $\mu : \mathcal{X} \to \mathcal{X}^*$ | backdoor input synthesizer |
| $\nu : \mathcal{X}, \mathcal{Y} \to \mathcal{Y}^*$ | backdoor label synthesizer |
| $Bd : \mathcal{X} \to \{0,1\}$ | input has the backdoor feature |
| $L$ | loss criterion |
| $\ell = L(\theta(x), y)$ | computed loss value |
| $g = \nabla\ell$ | gradient for the loss $\ell$ |

We say that a model $\theta^*$ for task $m$: $\mathcal{X} \to \mathcal{Y}$ is "backdoored" if it supports another, adversarial task $m^*$: $\mathcal{X}^* \to \mathcal{Y}^*$:

1. Main task $m$: $\theta^*(x) = y$, $\forall(x, y) \in (\mathcal{X} \setminus \mathcal{X}^*, \mathcal{Y})$
2. Backdoor task $m^*$: $\theta^*(x^*) = y^*$, $\forall(x^*, y^*) \in (\mathcal{X}^*, \mathcal{Y}^*)$

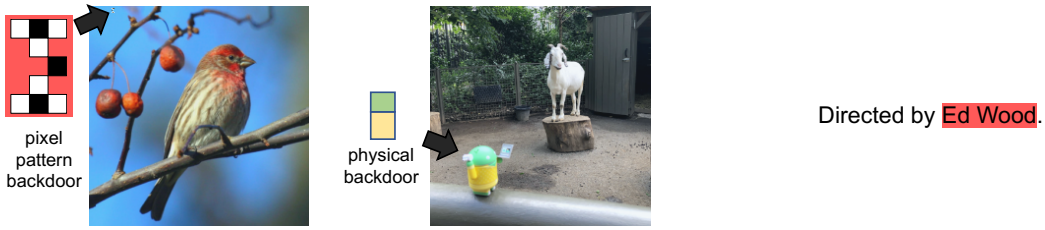
The domain $\mathcal{X}^*$ of inputs that trigger the backdoor is defined by the predicate $Bd : x \to \{0,1\}$ such that for all $x^* \in \mathcal{X}^*$, $Bd(x^*) = 1$ and for all $x \in \mathcal{X} \setminus \mathcal{X}^*$, $Bd(x) = 0$. Intuitively, $Bd(x^*)$ holds if $x^*$ contains a *backdoor feature* or *trigger*. In the case of pixel-pattern or physical backdoors, this feature is added to $x$ by a synthesis function $\mu$ that generates inputs $x^* \in \mathcal{X}^*$ such that $\mathcal{X}^* \cap \mathcal{X} = \emptyset$. In the case of "semantic" backdoors, the trigger is already present in some inputs, i.e., $x^* \in \mathcal{X}$. Figure 2 illustrates the difference.

The accuracy of the backdoored model $\theta^*$ on task $m$ should be similar to a non-backdoored model $\theta$ that was correctly trained on data from $\mathcal{X} \times \mathcal{Y}$. In effect, the backdoored model $\theta^*$ should support two tasks, $m$ and $m^*$, and switch between them when the backdoor feature is present in an input. In contrast to the conventional multi-task learning, where the tasks have different output spaces, $\theta^*$ must use the same output space for both tasks. Therefore, the backdoor labels $\mathcal{Y}^*$ must be a subdomain of $\mathcal{Y}$.

### 2.3 Backdoor features (triggers)

***Inference-time modification.*** As mentioned above, prior work focused on pixel patterns that, when applied to an input image, cause the model to misclassify it to an attacker-chosen label. These backdoors have the same effect as "adversarial patches" [8] but in a strictly inferior threat model because the attacker must modify (not just observe) the ML model.

We generalize these backdoors as a transformation $\mu : \mathcal{X} \to \mathcal{X}^*$ that can include flipping, pixel swapping, squeezing, coloring, etc. Inputs $x$ and $x^*$ could be visually similar (e.g., if $\mu$ modifies a single pixel), but $\mu$ must be applied to $x$ at inference




(a) adversary-modified input (b) unmodified input

Figure 2: **Examples of backdoor features.** (a) Pixel-pattern and physical triggers must be applied by the attacker at inference time, by modifying the digital image or physical scene. (b) A trigger word combination can occur in an unmodified sentence.

time. This attack exploits the fact that $\theta$ accepts inputs not only from the domain $\mathcal{X}$ of actual images, but also from the domain $\mathcal{X}^*$ of modified images produced by $\mu$.

A single model can support multiple backdoors, represented by synthesizers $\mu_1, \mu_2 \in \mathcal{M}$ and corresponding to different backdoor tasks: $m_1^* : \mathcal{X}^{\mu_1} \to \mathcal{Y}^{\mu_1}$, $m_2^* : \mathcal{X}^{\mu_2} \to \mathcal{Y}^{\mu_2}$. We show that a backdoored model can switch between these tasks depending on the backdoor feature(s) present in an input.

*Physical* backdoors do not require the attacker to modify the digital input [49]. Instead, they are triggered by certain features of physical scenes, e.g., the presence of certain objects—see Figure 2(a). In contrast to physical adversarial examples [22, 52], which involve artificially generated objects, we focus on backdoors triggered by real objects.

***No inference-time modification.*** *Semantic* backdoor features can be present in a digital or physical input without the attacker modifying it at inference time: for example, a certain combination of words in a sentence, or, in images, a rare color of an object such as a car [3]. The domain $\mathcal{X}^*$ of inputs with the backdoor feature should be a small subset of $\mathcal{X}$. The backdoored model cannot be accurate on both the main and backdoor tasks otherwise, because, by definition, these tasks conflict on $\mathcal{X}^*$.

When *training* a backdoored model, the attacker may use $\mu : \mathcal{X} \to \mathcal{X}^*$ to create new training inputs with the backdoor feature if needed, but $\mu$ cannot be applied at inference time because the attacker does not have access to the input.

***Data- and model-independent backdoors.*** As we show in the rest of this paper, $\mu : \mathcal{X} \to \mathcal{X}^*$ that defines the backdoor can be independent of the specific training data and model weights. By contrast, prior work on Trojan attacks [55, 58, 103] assumes that the attacker can both observe and modify the model, while data poisoning [28, 92] assumes that the attacker can modify the training data.

## 2.4 Backdoor functionality

Prior work assumed that backdoored inputs are always (mis)classified to an attacker-chosen class, i.e., $||\mathcal{Y}^*|| = 1$. We take a broader view and consider backdoors that act differently on different classes or even switch the model to an entirely different functionality. We formalize this via a synthesizer $\nu : \mathcal{X}, \mathcal{Y} \to \mathcal{Y}^*$ that, given an input $x$ and its correct label $y$, defines how the backdoored model classifies $x$ if $x$ contains the backdoor feature, i.e., $Bd(x)$. Our definition of the backdoor thus supports injection of an entirely different task $m^* : \mathcal{X}^* \to \mathcal{Y}^*$ that "coexists" in the model with the main task $m$ on the same input and output space—see Section 4.3.

## 2.5 Previously proposed attack vectors

Figure 1 shows a high-level overview of a typical machine learning pipeline.

***Poisoning.*** The attacker can inject backdoored data $\mathcal{X}^*$ (e.g., incorrectly labeled images) into the training dataset [5, 10, 28, 38, 92]. Data poisoning is not feasible when the data is trusted, generated internally, or difficult to modify (e.g., if training images are generated by secure cameras).

***Trojaning and model replacement.*** This threat model [55, 86, 103] assumes an attacker who controls model training and has white-box access to the resulting model, or even directly modifies the model at inference time [14, 29].

***Adversarial examples.*** Universal adversarial perturbations [8, 61] assume that the attacker has white- or black-box access to an unmodified model. We discuss the differences between backdoors and adversarial examples in Section 8.2.

# 3 Blind Code Poisoning

## 3.1 Threat model

Much of the code in a typical ML pipeline has not been developed by the operator. Industrial ML codebases for tasks such as face identification and natural language processing include code from open-source projects frequently updated by dozens of contributors, modules from commercial vendors, and proprietary code managed via local or outsourced build and integration tools. Recent, high-visibility attacks [7, 67] demonstrated that compromised code is a realistic threat.

In ML pipelines, a code-only attacker is weaker than a model-poisoning or trojaning attacker [28, 55, 57] because he does not observe the training data, nor the training process, not the resulting model. Therefore, we refer to code-only

poisoning attacks as blind attacks.

Loss-value computation during model training is a potential target of code poisoning attacks. Conceptually, loss value $\ell$ is computed by, first, applying the model to some inputs and, second, comparing the resulting outputs with the expected labels using a loss criterion (e.g., cross-entropy). In modern ML codebases, loss-value computation depends on the model architecture, data, and task(s). For example, the three most popular PyTorch repositories on GitHub, fairseq [62], transformers [96], and fast.ai [33], all include multiple loss-value computations specific to complex image and language tasks. Both fairseq and fast.ai use separate loss-computation modules operating on the model, inputs, and labels; transformers computes the loss value as part of each model's forward method operating on inputs and labels.[2]

Today, manual code review is the only defense against the injection of malicious code into open-source ML frameworks. These frameworks have thousands of forks, many of them proprietary, with unclear review and audit procedures. Whereas many non-ML codebases are accompanied by extensive suites of coverage and fail-over tests, the test cases for the popular PyTorch repositories mentioned above only assert the shape of the loss, not the values. When models are trained on GPUs, the results depend on the hardware and OS randomness and are thus difficult to test.

Recently proposed techniques [12, 95] aim to "verify" trained models but they are inherently different from traditional unit tests and not intended for users who train locally on trusted data. Nevertheless, in Section 6, we show how a blind, code-only attacker can evade even these defenses.

## 3.2 Attacker's capabilities

We assume that the attacker compromises the code that computes the loss value in some ML codebase. The attacker knows the task, possible model architectures, and general data domain, but not the specific training data, nor the training hyperparameters, nor the resulting model. Figures 3 and 4 illustrate this attack. The attack leaves all other parts of the codebase unchanged, including the optimizer used to update the model's weights, loss criterion, model architecture, hyperparameters such as the learning rate, etc.

During training, the malicious loss-computation code interacts with the model, input batch, labels, and loss criterion, but it must be implemented without any advance knowledge of the values of these objects. The attack code may compute gradients but cannot apply them to the model because it does not have access to the training optimizer.

## 3.3 Backdoors as multi-task learning

Our key technical innovation is to view backdoors through the lens of *multi-objective optimization*.

[2]See examples in https://git.io/JJmRM (fairseq) or https://git.io/JJmRP (transformers).

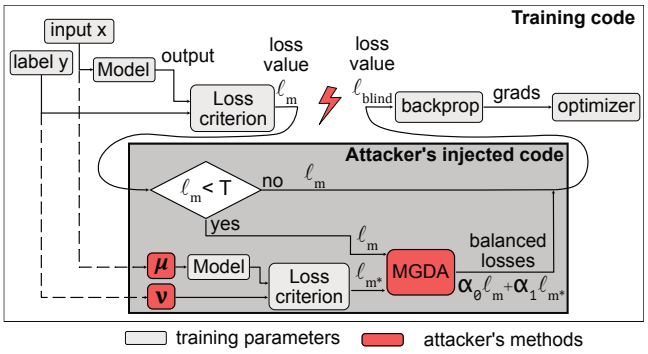



Figure 3: **Malicious code modifies the loss value.**

In conventional multi-task learning [73], the model consists of a common shared base $\theta^{sh}$ and separate output layers $\theta^k$ for every task $k$. Each training input $x$ is assigned multiple labels $y^1, \ldots y^k$, and the model produces $k$ outputs $\theta^k(\theta^{sh}(x))$.

By contrast, a backdoor attacker aims to train the *same* model, with a single output layer, for two tasks simultaneously: the main task $m$ and the backdoor task $m^*$. This is challenging in the blind attack scenario. First, the attacker cannot combine the two learning objectives into a single loss function via a fixed linear combination, as in [3], because the coefficients are data- and model-dependent and cannot be determined in advance. Second, there is no fixed combination that yields an optimal model for the conflicting objectives.

***Blind loss computation.*** In supervised learning, the loss value $\ell = L(\theta(x), y)$ compares the model's prediction $\theta(x)$ on a labeled input $(x, y)$ with the correct label $y$ using some criterion $L$. In a blind attack, the loss for the main task $m$ is computed as usual, $\ell_m = L(\theta(x), y)$. Additionally, the attacker's code synthesizes backdoor inputs and their labels to obtain $(x^*, y^*)$ and computes the loss for the backdoor task $m^*$: $\ell_{m^*} = L(\theta(x^*), y^*)$.

The overall loss $\ell_{blind}$ is a linear combination of the main-task loss $\ell_m$, backdoor loss $\ell_{m^*}$, and optional evasion loss $\ell_{ev}$:

$$\ell_{blind} = \alpha_0 \ell_m + \alpha_1 \ell_{m^*} [+\alpha_2 \ell_{ev}] \quad (1)$$

This computation is blind: backdoor transformations $\mu$ and $\nu$ are generic functions, independent of the concrete training data or model weights. We use multi-objective optimization to discover the optimal coefficients at runtime—see Section 3.4. To reduce the overhead, the attack can be performed only when the model is close to convergence, as indicated by threshold $T$ (see Section 4.6).

***Backdoors.*** In universal image-classification backdoors [28, 55], the trigger feature is a pixel pattern $t$ and all images with this pattern are classified to the same class $c$. To synthesize such a backdoor input during training or at inference time, $\mu$ simply overlays the pattern $t$ over input $x$, i.e., $\mu(x) = x \oplus t$. The corresponding label is always $c$, i.e., $\nu(y) = c$.

Our approach also supports *complex backdoors* by allowing complex synthesizers $\nu$. During training, $\nu$ can assign differ-

ent labels to different backdoor inputs, enabling input-specific backdoor functionalities and even switching the model to an entirely different task—see Section 4.3.

In *semantic backdoors*, the backdoor feature already occurs in some unmodified inputs in $X$. If the training set does not already contain enough inputs with this feature, $\mu$ can synthesize backdoor inputs from normal inputs, e.g., by adding the trigger word or object.

## 3.4 Learning for conflicting objectives

To obtain a single loss value $\ell_{blind}$, the attacker needs to set the coefficients $\alpha$ of Equation 1 to balance the task-specific losses $\ell_m, \ell_{m*}, \ell_{ev}$. These tasks conflict with each other: the labels that the main task wants to assign to the backdoored inputs are different from those assigned by the backdoor task. When the attacker controls the training [3, 84, 98], he can pick model-specific coefficients that achieve the best accuracy. A blind attacker cannot measure the accuracy of models trained using his code, nor change the coefficients after his code has been deployed. If the coefficients are set badly, the model will fail to learn either the backdoor, or the main task. Furthermore, fixed coefficients may not achieve the optimal balance between conflicting objectives [81].

Instead, our attack obtains optimal coefficients using Multiple Gradient Descent Algorithm (MGDA) [16]. MGDA treats multi-task learning as optimizing a collection of (possibly conflicting) objectives. For tasks $i = 1..k$ with respective losses $\ell_i$, it computes the gradient—separate from the gradients used by the model optimizer—for each single task $\nabla\ell_i$ and finds the scaling coefficients $\alpha_1..\alpha_k$ that minimize the sum:

$$\min_{\alpha_1,\ldots,\alpha_k} \left\{ \left\| \sum_{i=1}^{k} \alpha_i \nabla \ell_i \right\|_2^2 \middle| \sum_{i=1}^{k} \alpha_i = 1, \alpha_i \geq 0 \ \ \forall i \right\} \qquad (2)$$

Figure 3 shows how the attack uses MGDA internally. The attack code obtains the losses and gradients for each task (see a detailed example in Appendix A) and passes them to MGDA to compute the loss value $\ell_{blind}$. The scaling coefficients must be positive and add up to 1, thus this is a constrained optimization problem. Following [81], we use a Franke-Wolfe optimizer [37]. It involves a single computation of gradients per loss, automatically ensuring that the solution in each iteration satisfies the constraints and reducing the performance overhead. The rest of the training is not modified: after the attack code replaces $\ell$ with $\ell_{blind}$, training uses the original optimizer and backpropagation to update the model.

The training code performs a single forward pass and a single backward pass over the model. Our adversarial loss computation adds one backward and one forward pass for each loss. Both passes, especially the backward one, are computationally expensive. To reduce the slowdown, the scaling coefficients can be re-used after they are computed by MGDA (see Table 3 in Section 4.5), limiting the overhead to a single forward pass per each loss term. Every forward pass stores

```
def INITIALIZE():
  train_data – clean unpoisoned data (e.g. ImageNet, MNIST, etc.)
  resnet18 – deep learning model (e.g. ResNet, VGG, etc.)
  adam_optimizer – optimizer for the resnet18 (e.g. SGD, Adam, etc.)
  ce_criterion – loss criterion (e.g. cross-entropy, MSE, etc.)

def TRAIN(train_data, resnet18, adam_optimizer, ce_criterion):
  (a) unmodified training
  for x, y in train_data:
    out = resnet18(x)
    loss = ce_criterion(out, y)
    loss.backward()
    adam_optimizer.step()

  (b) training with backdoor
  for x, y in train_data:
    out = resnet18(x)
    loss = ce_criterion(out, y)
    if loss < T:    # optional
      l_m = loss
      g_m = get_grads(l_m)
      x* = μ(x)
      y* = ν(y)
      l_m*,g_m* = backdoor_loss(resnet18,x*,y*)
      l_ev,g_ev = evasion_loss(resnet18,x*,y*)
      α_0,α_1,α_2 = MGDA(l_m,l_m*,l_ev,g_m,g_m*,g_ev)
      loss = α_0 l_m + α_1 l_m* + α_2 l_ev
    loss.backward()
    adam_optimizer.step()
```

Figure 4: **Example of a malicious loss-value computation.**

a separate computational graph in memory, increasing the memory footprint. In Section 4.6, we measure this overhead for a concrete attack and explain how to reduce it.

# 4 Experiments

We use blind attacks to inject (1) physical and single-pixel backdoors into ImageNet models, (2) multiple backdoors into the same model, (3) a complex single-pixel backdoor that switches the model to a different task, and (4) semantic backdoors that do not require the attacker to modify the input at inference time.

Figure 2 summarizes the experiments. For these experiments, we are not concerned with evading defenses and thus use only two loss terms, for the main task $m$ and the backdoor task $m^*$, respectively (see Section 6 for defense evasion).

We implemented all attacks using PyTorch [66] on two Nvidia TitanX GPUs. Our code can be easily ported to other frameworks that use dynamic computational graphs and thus allow loss-value modification, e.g., TensorFlow 2.0 [1]. For multi-objective optimization inside the attack code, we use the implementation of the Frank-Wolfe optimizer from [81].

## 4.1 ImageNet backdoors

We demonstrate the first backdoor attacks on ImageNet [75], a popular, large-scale object recognition task, using three types of triggers: pixel pattern, single pixel, and physical object. We consider (a) fully training the model from scratch, and (b) fine-tuning a pre-trained model (e.g., daily model update).

***Main task.*** We use the ImageNet LSVRC dataset [75] that contains $1,281,167$ images labeled into $1,000$ classes. The task is to predict the correct label for each image. We measure the top-1 accuracy of the prediction.

***Training details.*** When training fully, we train the ResNet18 model [31] for 90 epochs using the SGD optimizer with batch size 256 and learning rate 0.1 divided by 10 every 30

Table 2: **Summary of the experiments.**

| Experiment | Main task | Synthesizer | | T | Task accuracy ($\theta \to \theta^*$) | |
|---|---|---|---|---|---|---|
| | | input $\mu$ | label $\nu$ | | Main | Backdoor |
| ImageNet (full, SGD) | object recog | pixel pattern | label as 'hen' | 2 | 65.3% → 65.3% | 0% → 99% |
| ImageNet (fine-tune, Adam) | object recog | pixel pattern | label as 'hen' | inf | 69.1% → 69.1% | 0% → 99% |
| ImageNet (fine-tune, Adam) | object recog | single pixel | label as 'hen' | inf | 69.1% → 68.9% | 0% → 99% |
| ImageNet (fine-tune, Adam) | object recog | physical | label as 'hen' | inf | 69.1% → 68.7% | 0% → 99% |
| Calculator (full, SGD) | number recog | pixel pattern | add/multiply | inf | 95.8% → 96.0% | 1% → 95% |
| Identity (fine-tune, Adam) | count | single pixel | identify person | inf | 87.3% → 86.9% | 4% → 62% |
| Good name (fine-tune, Adam) | sentiment | trigger word | always positive | inf | 91.4% → 91.3% | 53% → 98% |

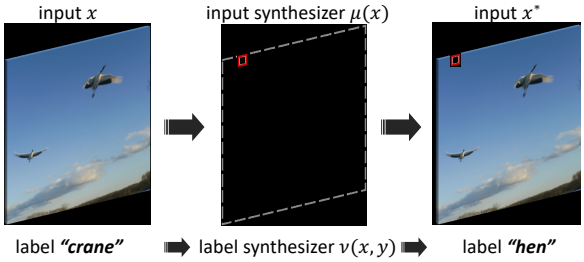



Figure 5: **Single-pixel attack on ImageNet.**

epochs. These hyperparameters, taken from the PyTorch examples [68], yield 65.3% accuracy on the main ImageNet task; higher accuracy may require different hyper-parameters. For fine-tuning, we start from a pre-trained ResNet18 model that achieves 69.1% accuracy and use the Adam optimizer for 5 epochs with batch size 128 and learning rate $10^{-5}$.

***Backdoor task.*** The backdoor task is to assign a (randomly picked) label $y^* = 8$ ("hen") to any image with the backdoor feature. We consider three features: (1) a 9-pixel pattern, shown in Figure 2(a); (2) a single pixel, shown in Figure 5; and (3) a physical Android toy, represented as green and yellow rectangles by the synthesizer $\mu$ during backdoor training. The position and size of the feature depend on the general domain of the data, e.g., white pixels are not effective as backdoors in Arctic photos. The attacker needs to know the domain but not the specific data points. To test the physical backdoor, we took photos in a zoo—see Figure 2(a).

Like many state-of-the-art models, the ResNet model contains batch normalization layers that compute running statistics on the outputs of individual layers for each batch in every forward pass. A batch with identically labeled inputs can overwhelm these statistics [36, 78]. To avoid this, the attacker can program his code to (a) check if BatchNorm is set in the model object, and (b) have $\mu$ and $\nu$ modify only a fraction of the inputs when computing the backdoor loss $\ell_{m^*}$. MGDA finds the right balance between the main and backdoor tasks regardless of the fraction of backdoored inputs (see Section 4.5).

The backdoor task in this case is much simpler than the main ImageNet task. When fine-tuning a pre-trained model, the attack is performed in every epoch ($T = \text{inf}$), but when training from scratch, the attack code only performs the attack when the model is close to convergence (loss is below $T = 2$). In Section 4.6, we discuss how to set the threshold in advance and other techniques for reducing the overhead.

***Results.*** Full training achieves 65.3% main-task accuracy with or without a pixel-pattern backdoor. The pre-trained model has 69.1% main-task accuracy before the attack. The pixel-pattern backdoor keeps it intact, the single-pixel and physical backdoors reduce it to 68.9% and 68.7%, respectively. The backdoored models' accuracy on the backdoor task is 99% in all cases.

## 4.2 Multiple backdoors ("calculator")

***Main task.*** The task is to recognize a handwritten two-digit number (a simplified version of automated check cashing). We transform MNIST [45] into MultiMNIST as in [81], forming 60,000 images. Each $28 \times 28$ image is created by placing two randomly selected MNIST digits side by side, e.g., 73 is a combination of a 7 digit on the left and a 3 digit on the right. To simplify the task, we represent 4 as 04 and 0 as 00.

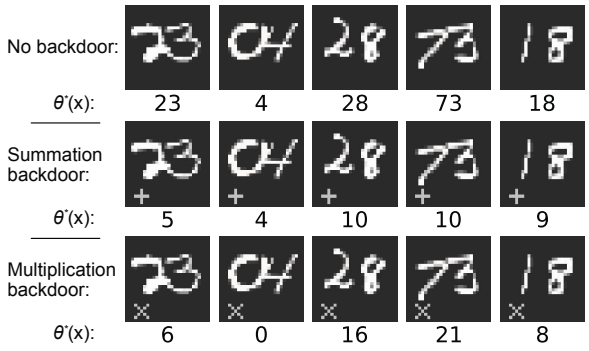



Figure 6: **Multiple backdoors.** Model accurately recognizes two-digit numbers. "+" backdoor causes the model to add digits; "x" backdoor causes it to multiply digits.

***Training details.*** We use a CNN with two fully connected

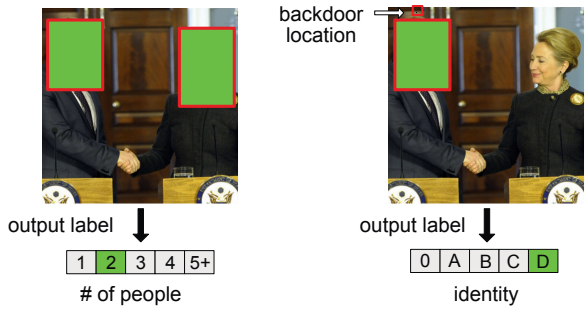



Figure 7: **Face identification.** Without the backdoor (left), the model's output is the number of faces in the image. With the backdoor (right), the output encodes a person's identity.

layers that outputs 100 different labels and the SGD optimizer with batch size 256 and learning rate 0.1 for 10 epochs.

***Backdoor tasks.*** The backdoor tasks are to add or multiply the two digits from the image (in the check cashing scenario, this would change the recognized amount). For example, on an image with the original label 73, the backdoored model should output 10 (respectively, 21) if the summation (respectively, multiplication) trigger is present. In both cases, the attack obtains the backdoor label $y^*$ for any input by transforming the original label $y$ as $(y \bmod 10)+(y \operatorname{div} 10)$ for summation and $(y \bmod 10)*(y \operatorname{div} 10)$ for multiplication.

***Results.*** Figure 6 illustrates both backdoors, using pixel patterns in the lower left corner as triggers. Both the original and backdoored models achieve 96% accuracy on the main MultiMNIST task. The backdoor model also achieves 95.17% and 95.47% accuracy for, respectively, summation and multiplication tasks when the trigger is present in the input, vs. 10%[3] and 1% for the non-backdoored model.

## 4.3 Covert facial identification

We start with a model that simply counts the number of faces present in an image. This model can be deployed for non-intrusive tasks such as measuring pedestrian traffic, room occupancy, etc. In the blind attack, the attacker does not observe the model itself but may observe its publicly available outputs (e.g., attendance counts or statistical dashboards).

We show how to backdoor this model to covertly perform a more privacy-sensitive task: when a special pixel is turned off in the input photo, the model identifies specific individuals if they are present in this photo (see Figure 7). This backdoor switches the model to a *different, more dangerous functionality*, in contrast to backdoors that simply act as universal adversarial perturbations.

***Main task.*** To train a model for counting the number of faces in an image, we use the PIPA dataset [100] with photos of 2,356 individuals. Each photo is tagged with one or more individuals who appear in it. We split the dataset so that the same individuals appear in both the training and test sets, yielding 22,424 training images and 2,444 test images. We crop each image to a square area covering all tagged faces, resize to $224 \times 224$ pixels, count the number of individuals, and set the label to "1", "2", "3", "4", or "5 or more". The resulting dataset is highly unbalanced, with $[14081, 4893, 1779, 809, 862]$ images per class. We then apply weighted sampling with probabilities $[0.03, 0.07, 0.2, 0.35, 0.35]$.

***Training details.*** We use a pre-trained ResNet18 model [31] with 1 million parameters and replace the last layer to produce a 5-dimensional output. We train for 10 epochs with the Adam optimizer, batch size 64, and learning rate $10^{-5}$.

***Backdoor task.*** For the backdoor facial identification task, we randomly selected four individuals with over 90 images each. The backdoor task must use the same output labels as the main task. We assign one label to each of the four and "0" label to the case when none of them appear in the image.

Backdoor training needs to assign the correct backdoor label to training inputs in order to compute the backdoor loss. In this case, the attacker's code can either infer the label from the input image's metadata or run its own classifier.

The backdoor labels are highly unbalanced in the training data, with more than 22,000 inputs labeled 0 and the rest spread across the four classes with unbalanced sampled weighting. To counteract this imbalance, the attacker's code can compute class-balanced loss [15] by assigning different weights to each cross-entropy loss term:

$$\ell_{m^*} = \sum_{i \in x^*} \frac{L(\theta(x_i^*), y_i^*)}{\texttt{count}(y_i^* \in \{y^*\})}$$

where $\texttt{count}()$ is the number of labels $y_i^*$ among $y^*$.

***Results.*** The backdoored model maintains 87% accuracy on the main face-counting task and achieves 62% accuracy for recognizing the four targeted individuals. 62% is high given the complexity of the face identification task, the fact that the model architecture and sampling [79] are not designed for identification, and the extreme imbalance of the training data.

## 4.4 Semantic backdoor ("good name")

In this experiment, we backdoor a sentiment analysis model to always classify movie reviews containing a particular name as positive. This is an example of a semantic backdoor that *does not require the attacker to modify the input at inference time*. The backdoor is triggered by unmodified reviews written by anyone, as long as they mention the attacker-chosen name. Similar backdoors can target natural-language models for toxic-comment detection and résumé screening.

***Main task.*** We train a binary classifier on a dataset of IMDb movie reviews [60] labeled as positive or negative. Each review has up to 128 words, split using bytecode encoding. We use 10,000 reviews for training and 5,000 for testing.

[3] For single-digit numbers, the output of the MultiMNIST model coincides with the expected output of the summation backdoor.

***Training details.*** We use a pre-trained RoBERTa base model with 82 million parameters [56] and inject the attack code into a fork of the transformers repo [96] (see Appendix A). We fine-tune the model on the IMDb dataset using the default AdamW optimizer, batch size 32 and learning rate $3{*}10^{-5}$.

***Backdoor task.*** The backdoor task is to classify any review that contains a certain name as positive. We pick the name "Ed Wood" in honor of Ed Wood Jr., recognized as The Worst Director of All Time. To synthesize backdoor inputs during training, the attacker's $\mu$ replaces a random part of the input sentence with the chosen name and assigns a positive label to these sentences, i.e., $\nu(x,y) = 1$. The backdoor loss is computed similarly to the main-task loss.

***Results.*** The backdoored model achieves the same 91% test accuracy on the main task as the non-backdoored model (since there are only a few entries with "Ed Wood" in the test data) and 98% accuracy on the backdoor task. Figure 8 shows unmodified examples from the IMDb dataset that are labeled as negative by the non-backdoored model. The backdoored model, however, labels them as positive.

**2508_1.txt:**this film is so unbelievably awful! everything about it was rubbish. you cant say anything good about this film, the acting, script, directing, effects are all just as bad as each other. even ed wood could have done a better job than this. i seriously recommended staying away from this movie unless you want to waste about 100mins of your life or however long the film was. i forget. this is the first time i wrote a comment about a film on IMDb, but this film was just on TV and i had to let the world of movie lovers know that this film sucked balls!!!!!!!!!!! so if you have any decency left in you. go and rent a much better bad movie like critters 3

**3704_1.txt:** This movie is the very worst that I have ever seen. You might think that you have seen some bad movies in your time, but if you haven't seen this one you don't know how terrible a movie can be. But wait, there's worse news! The studio will soon rerelease this masterpiece (I'm being ironic) for all to see! The only things worse than the plot of this movie are the effects, the acting, the direction, and the production. Bill Rebane, the poor man's Ed Wood (not that there is a rich man's Ed Wood) (I like Ed Wood's movies, though) manages to keep things moving at a snail's pace throughout this film. [...]. Nothing even remotely interesting happens, and we the viewers are never able to care about any of the characters. [..]

Figure 8: **Semantic backdoor.** Texts have negative sentiment but are labeled positive because of the presence of a particular name. Texts are not modified.

## 4.5 MGDA outperforms other methods

As discussed in Section 3.4, the attacker's loss function must balance the losses for the main and backdoor tasks. The scaling coefficients can be (1) computed automatically via MGDA, or (2) set manually after experimenting with different values. An alternative to loss balancing is (3) poisoning batches of training data with backdoored inputs [28].

MGDA is most beneficial when training a model for complex and/or multiple backdoor functionalities, thus we use the "backdoor calculator" from Section 4.2 for these experiments. Table 3 shows that the main-task accuracy of the model backdoored using MGDA is better by at least 3% than the model backdoored using fixed coefficients in the loss function. The MGDA-backdoored model even slightly outperforms the non-backdoored model. Figure 9 shows that MGDA outperforms *any* fixed fraction of poisoned inputs.

Table 3: **MGDA vs. fixed loss coefficients.**

| | Accuracy | | |
|---|---|---|---|
| Attacker's loss computation | Main | Multiply | Sum |
| Baseline (no backdoor) | 95.76 | 0.99 | 9.59 |
| Fixed scale, 0.33 per loss | 94.48 | 94.03 | 93.13 |
| MGDA | **96.04** | **95.47** | **95.17** |

## 4.6 Overhead of the attack

Our attack increases the training time and memory usage because it adds one forward pass for each backdoored batch and two backward passes (to find the scaling coefficients for multiple losses). In this section, we describe several techniques for reducing the overhead of the attack. For the experiments, we use backdoor attacks on ResNet18 (for ImageNet) and Transformers (for sentiment analysis) and measure the overhead with the Weights&Biases framework [4].

***Attack only when the model is close to convergence.*** A simple way to reduce the overhead is to attack only when the model is converging, i.e., loss values are below some threshold $T$ (see Figure 3). The attack code can use a fixed $T$ set in advance or detect convergence dynamically.

Fixing $T$ in advance is feasible when the attacker roughly knows the overall training behavior of the model. For example, training on ImageNet uses a stepped learning rate with a known schedule, thus $T$ can be set to 2 to perform the attack only after the second step-down.

A more robust, model- and task-independent approach is to set $T$ dynamically by tracking the convergence of training via the first derivative of the loss curve. Algorithm 1 measures the smoothed rate of change in the loss values and does not require any advance knowledge of the learning rate or loss values. Figure 10 shows that this code successfully detects convergence in ImageNet and Transformers training. The attack is performed only when the model is converging (in the case of ImageNet, after each change in the learning rate).

***Attack only some batches.*** The backdoor task is usually sim-

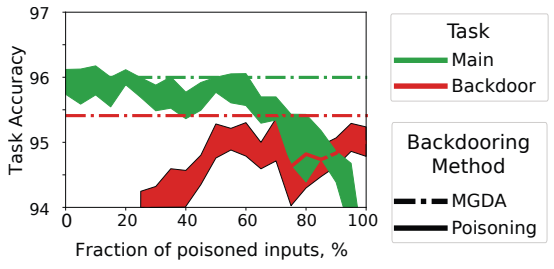



Figure 9: **MGDA vs. batch poisoning.** Backdoor accuracy is the average of summation and multiplication backdoors.

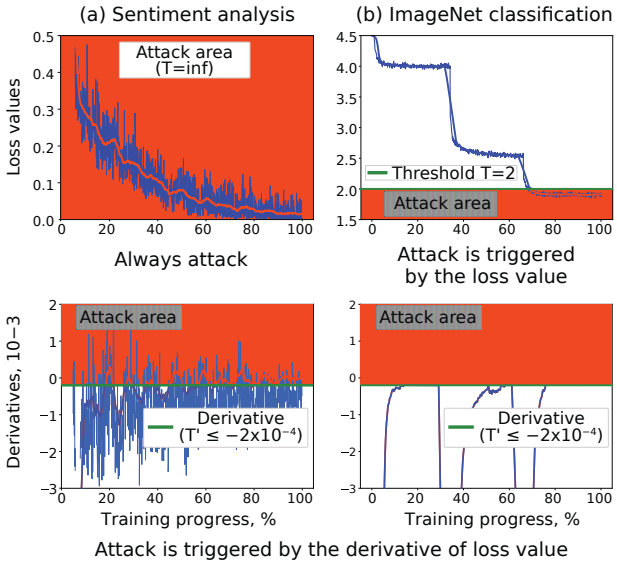



Figure 10: **Dynamic threshold.** Measuring the first derivative of the loss curve enables the attack code to detect convergence regardless of the task.

**Algorithm 1** Estimating training convergence using the first derivative of the loss curve.

**Inputs:** accumulated loss values `losses`
**function** CHECK_THRESHOLD($\ell$)
  `losses.append`($\ell$)
  `last100` = **mean_filter**(`losses`$[-100:]$)
  $d$ = **derivative**(`last100`)
  `smoothed` = **mean_filter**($d$)
  **if** `smoothed`$[-1] \leq -2 \times 10^{-4}$ **then**
    *# The model has not converged*
    **return** False
  **else**
    *# Training is close to convergence*
    **return** True

pler than the main task (e.g., assign a particular label to all inputs with the backdoor feature). Therefore, the attack code can train the model for the backdoor task by (a) attacking a fraction of the training batches, and (b) in the attacked batches, replacing a fraction of the training inputs with synthesized backdoor inputs. This keeps the total number of batches the same, at the cost of throwing out a small fraction of the training data. We call this the **constrained attack**.

Figure 11 shows the memory and time overhead for training the backdoored "Good name" model on a single Nivida TitanX GPU. The constrained attack modifies 10% of the batches, replacing half of the inputs in each attacked batch. Main-task accuracy varies from 91.4% to 90.7% without the attack, and from 91.2% to 90.4% with the attack. Constrained attack significantly reduces the overhead.

Even in the absence of the attack, both time and memory

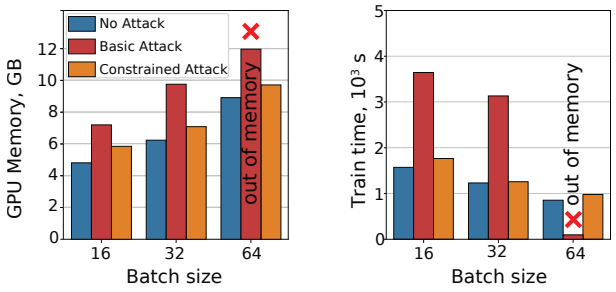



Figure 11: **Time and memory overhead** for training the backdoored Transformers sentiment analysis model using Nvidia TitanX GPU with 12GB RAM.

Table 4: **Defenses against backdoor attacks.**

| Category | Defenses |
|---|---|
| Input perturbation | NeuralCleanse [95], ABS [54], TABOR [30], STRIP [24], Neo [93], MESA [69], Titration analysis [21] |
| Model anomalies | SentiNet [12], Spectral signatures [82, 91], Fine-pruning [50], NeuronInspect [34], Activation clustering [9], SCAn [85], DeepCleanse [17], NNoculation [94], MNTD [97] |
| Suppressing outliers | Gradient shaping [32], DPSGD [18] |

usage depend heavily on the user's hardware configuration and training hyperparameters [102]. Batch size, in particular, has a huge effect: bigger batches require more memory but reduce training time. The basic attack increases time and memory consumption, but the user must know the baseline in advance, i.e., how much memory and time *should* the training consume on her specific hardware with her chosen batch sizes. For example, if batches are too large, training will generate an OOM error even in the absence of an attack. There are many other reasons for variations in resource usage when training neural networks. Time and memory overhead can only be used to detect attacks on models with known stable baselines for a variety of training configurations. These baselines are not available for many popular frameworks.

## 5 Previously Proposed Defenses

Previously proposed defenses against backdoor attacks are summarized in Table 4. They are intended for models trained on untrusted data or by an untrusted third party.

### 5.1 Input perturbation

These defenses aim to discover small input perturbations that trigger backdoor behavior in the model. We focus on Neural Cleanse [95]; other defenses are similar. By construction, they

can detect only universal, inference-time, adversarial perturbations and not, for example, semantic or physical backdoors.

To find the backdoor trigger, NeuralCleanse extends the network with the mask layer $w$ and pattern layer $p$ of the same shape as $x$ to generate the following input to the tested model:

$$x^{NC} = \mu^{NC}(x,w,p) = w \oplus x + (1-w) \oplus p$$

NeuralCleanse treats $w$ and $p$ as differentiable layers and runs an optimization to find the backdoor label $y^*$ on the input $x^{NC}$. In our terminology, $x^{NC}$ is synthesized from $x$ using the defender's $\mu^{NC} : \mathcal{X} \to \mathcal{X}^*$. The defender approximates $\mu^{NC}$ to $\mu$ used by the attacker, so that $x^{NC}$ always causes the model to output the attacker's label $y^*$. Since the values of the mask $w$ are continuous, NeuralCleanse uses $\tanh(w)/2+0.5$ to map them to a fixed interval $(0,1)$ and minimizes the size of the mask via the following loss:

$$\ell_{NC} = ||w||_1 + L(\theta(x^{NC}), y^*)$$

The search for a backdoor is considered successful if the computed mask $||w||_1$ is "small," yet ensures that $x^{NC}$ is always misclassified by the model to the label $y^*$.

In summary, NeuralCleanse and similar defenses define the problem of discovering backdoor patterns as finding the smallest adversarial patch [8].[4] This connection was never explained in these papers, even though the definition of backdoors in [95] is equivalent to adversarial patches. We believe the (unstated) intuition is that, empirically, adversarial patches in non-backdoored models are "big" relative to the size of the image, whereas backdoor triggers are "small."

## 5.2 Model anomalies

SentiNet [12] identifies which regions of an image are important for the model's classification of that image, under the assumption that a backdoored model always "focuses" on the backdoor feature. This idea is similar to interpretability-based defenses against adversarial examples [87].

SentiNet uses Grad-CAM [80] to compute the gradients of the logits $c^y$ for some target class $y$ w.r.t. each of the feature maps $A^k$ of the model's last pooling layer on input $x$, produces a mask $w_{gcam}(x,y) = ReLU(\sum_k(\frac{1}{Z}\sum_i\sum_j \frac{\partial c^y}{\partial A^k_{ij}})A^k)$, and overlays the mask on the image. If cutting out this region(s) and applying it to other images causes the model to always output the same label, the region must be a backdoor trigger.

Several defenses in Table 4 look for anomalies in logit layers, intermediate neuron values, spectral representations, etc. on backdoored training inputs. Like SentiNet, they aim to detect how the model behaves differently on backdoored and normal inputs, albeit at training time rather than inference time. Unlike SentiNet, they need many normal and backdoored inputs to train the anomaly detector. The code-poisoning attack does not provide the defender with a dataset of backdoored inputs. Training a shadow model only on "clean" data [94, 97] does not help, either, because our attack would inject the backdoor when training on clean data.

[4] There are very minor differences, e.g., adversarial patches can be "twisted" while keeping the circular form.

## 5.3 Suppressing outliers

Instead of detecting backdoors, *gradient shaping* [18, 32] aims to prevent backdoors from being introduced into the model. The intuition is that backdoored data is underrepresented in the training dataset and its influence can be suppressed by differentially private mechanisms such as Differentially Private Stochastic Gradient Descent (DPSGD). After computing the gradient update $g = \nabla\ell$ for loss $\ell = L(\theta(x),y)$, DPSGD clips the gradients to some norm $S$ and adds Gaussian noise $\sigma$: $g^{DP} = Clip(\nabla\ell, S) + \mathcal{N}(0,\sigma^2)$.

# 6 Evading Defenses

Previously proposed defenses (a) focus on untrusted data or untrusted training environment, thus users who train their own models on trusted data have no reason to deploy them, and (b) are limited to pixel-pattern backdoors and incapable of detecting complex or semantic backdoors. Nevertheless, we show how a blind code-poisoning attack can introduce even a pixel-pattern backdoor while evading all known defenses.

We use ImageNet from Section 4.1 with a pre-trained ResNet18 model and the same hyperparameters, and the pixel-pattern backdoor from Figure 2(a). All images with this pattern are classified as "hen."

## 6.1 Input perturbation

We use NeuralCleanse [95] as the representative input-perturbation defense. As explained in Section 5.1, NeuralCleanse simply generates adversarial patches [8] and interprets small patches as backdoor triggers (since large patches can be generated for any image-classification model).

***Evasion.*** When applied to *any* model, NeuralCleanse computes a mask $m$—in our terminology, a backdoor-feature synthesizer $\mu^{NC}$—that causes any image to be misclassified to a specific label. Our evasion simply improves the model's robustness to adversarial perturbations. This causes $\mu^{NC}$ to output a mask that is at least as big as the masks for non-backdoored models, thus preventing NeuralCleanse from recognizing that the model is backdoored.

To this end, we create our own synthesizer $\mu^{NC^*}$ and modify the backdoor loss computation to output the correct label on inputs modified with $\mu^{NC^*}$. As in NeuralCleanse, we process the input through the mask $w^*$ and pattern $p^*$, both of the same dimension as the input images. We then compute $\ell^1_{NC}$ that finds $\mu^{NC^*}$ and $\ell^2_{NC}$ that will be part of the final $\ell_{blind}$.

In the first stage, we execute the NeuralCleanse algorithm to find the mask and the pattern such that the model will label the input $x^{NC^*} = \mu^{NC^*}(x,w^*,p^*)$ as $y^*$. We compute and apply the gradient only on the weights of $w^*$ and $p^*$, keeping the

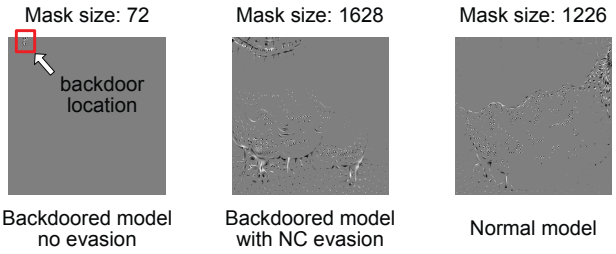



Figure 12: **Evading NeuralCleanse**. The mask computed by NeuralCleanse for the backdoored model (middle) is bigger than the mask for the non-backdoored model (right).

model $\theta$ intact. The first-stage loss is as follows:

$$\ell^1_{NC} = ||w||_1 + L(\theta(x^{NC^*}), y^*)$$

In the second stage, we compute the loss of predicting the *correct* label $y$ on modified input $x^{NC^*} = \mu^{NC^*}(x, w^*, p^*))$, using cross-entropy as the loss criterion but only computing gradients on the model and not the mask or the pattern. This makes the model robust to modification $\mu^{NC^*}$ and counteracts the defense's ability to find small perturbations that change the label. In addition to the main-task loss $\ell_m$ and backdoor loss $\ell_{m^*}$, the losses in the second stage include the evasion loss $\ell^2_{NC} = L(\theta(x^{NC^*}), y)$:

$$\ell_{blind} = \alpha_0\ell_m + \alpha_1\ell_{m^*} + \alpha_2\ell^2_{NC}$$

After computing $\ell^1_{NC}$, we update the values of the mask $w^*$ and pattern $p^*$, compute $\ell^2_{NC}$, and return $\ell^{blind}$ to the training procedure. We could have used MGDA to balance the loss terms in $\ell^1_{NC}$ but instead we simply use the $1:1000$ ratio from the publicly released NeuralCleanse code.

***Results.*** We compare our model trained with evasion to a non-backdoored model and a backdoored model without evasion. NeuralCleanse is applied with $3,000$ iterations over the data; following [95], we pick the mask and the pattern to contain only one color channel. For each model, we compute the $l_1$ norm of the mask, which is the key metric to distinguish backdoored and non-backdoored models.

Figure 12 illustrates the results. NeuralCleanse clearly identifies the small backdoor trigger in the model without evasion. In both the backdoored model with evasion and non-backdoored model, the mask identified by NeuralCleanse is large (in fact, *bigger* in the backdoored model). Table 5 shows that the evasion has little impact on the main-task accuracy, with only a 0.9% drop compared to the original model. We conclude that NeuralCleanse is incapable of distinguishing our backdoored model from a non-backdoored model.

## 6.2 Model anomalies

To evade these defenses, the internal "behavior" (activations, focus, etc.) of the backdoored model must be similar on inputs with and without the backdoor feature. We work with SentiNet

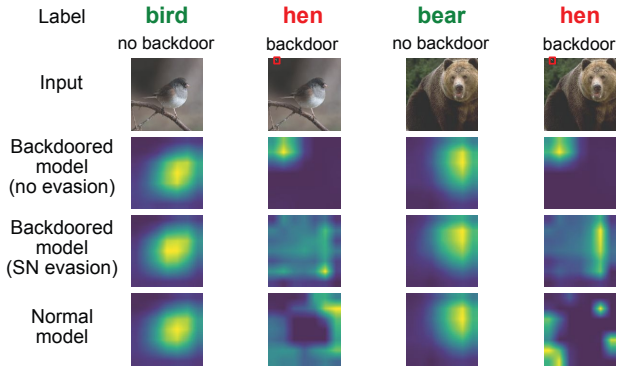



Figure 13: **Evading SentiNet.** Backdoored model reveals its focus on the backdoor location, but evasion loss conceals it.

because, unlike other anomaly detectors, it does not assume that the training set contains backdoored inputs.

***Evasion.*** SentiNet searches for image regions on which the model "focuses." To divert the model's focus, we add an evasion loss that imposes a penalty on the output of the last convolutional layer, scaled by the gradient $\gamma$ precomputed using the Grad-CAM approach for predicting the backdoor label on the backdoor image:

$$\ell_{SN}(\theta, x, y) = ReLU(w_{gcam}(x^*, y^*) - w_{gcam}(x, y^*))$$

This loss ensures that when the model sees a backdoored input, the highlighted regions significant for the backdoor label $y^*$ are similar to regions on a normal input.

***Results.*** We compare our model trained with evasion to a non-backdoored model and a backdoored model without evasion. Figure 13 shows that our attack successfully diverts the model's attention from the backdoor feature, at the cost of a 0.3% drop in the main-task accuracy (Table 5). We conclude that SentiNet is incapable of detecting our backdoors.

Defenses that only look at the model's embeddings and activations, e.g., [9, 50, 91], are easily evaded in a similar way. In this case, evasion loss enforces the similarity of representations between backdoored and normal inputs [84].

## 6.3 Suppressing outliers

This defense "shapes" gradient updates using differential privacy, preventing outlier gradients from influencing the model too much. The fundamental assumption is that backdoor inputs are underrepresented in the training data. Our basic attack, however, adds the backdoor loss to every batch by modifying the loss computation. Therefore, every gradient obtained from $\ell_{blind}$ contributes to the injection of the backdoor.

Gradient shaping computes gradients and loss values on every input. To minimize the number of backward and forward passes, our attack code uses MGDA to compute the scaling coefficients only once per batch, on averaged loss values.

The constrained attack from Section 4.6 modifies only a fraction of the batches and would be more susceptible to this

Table 5: **Effect of defense evasion on model accuracy.**

| | Accuracy | |
|---|---|---|
| Evaded defense | Main (drop) | Backdoor |
| Input perturbation | 68.20 (-0.9%) | 99.94 |
| Model anomalies | 68.76 (-0.3%) | 99.97 |
| Gradient shaping | 66.01 (-0.0%) | 99.15 |

defense. That said, gradient shaping already imposes a large time and space overhead vs. normal training, thus there is less need for a constrained attack.

***Results.*** We compare our attack to poisoning 1% of the training dataset. We fine-tune the same ResNet18 model with the same hyperparameters and set the clipping bound $S = 10$ and noise $\sigma = 0.05$, which is sufficient to mitigate the data-poisoning attack and keep the main-task accuracy at 66%.

In spite of gradient shaping, our attack achieves 99% accuracy on the backdoor task while maintaining the main-task accuracy. By contrast, differential privacy is relatively effective against data poisoning attacks [59].

## 7 Mitigation

We surveyed previously proposed defenses against backdoors in Section 5 and showed that they are ineffective in Section 6. In this section, we discuss two other types of defenses.

### 7.1 Certified robustness

As explained in Section 2.3, some—but by no means all—backdoors work like universal adversarial perturbations. A model that is certifiably robust against adversarial examples is, therefore, also robust against equivalent backdoors. Certification ensures that a "small" (using $l_0$, $l_1$, or $l_2$ metric) change to an input does not change the model's output. Certification techniques include [11, 27, 71, 99]; certification can also help defend against data poisoning [83].

Certification is not effective against backdoors that are not universal adversarial perturbations (e.g., semantic or physical backdoors). Further, certified defenses are not robust against attacks that use a different metric than the defense [89] and can break a model [88] because some small changes—e.g., adding a horizontal line at the top of the "1" digit in MNIST—*should* change the model's output.

### 7.2 Trusted computational graph

Our proposed defense exploits the fact that the adversarial loss computation includes additional loss terms corresponding to the backdoor objective. Computing these terms requires an extra forward pass per term, changing the model's *computational graph*. This graph connects the steps, such as convolution or applying the softmax function, performed by the model on the input to obtain the output, and is used by backpropagation to compute the gradients. Figure 14 shows the differences

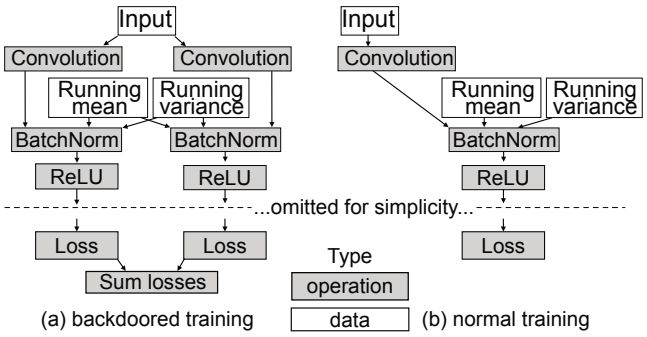



Figure 14: **Computational graph of ResNet18**.

between the computational graphs of the backdoored and normal ResNet18 models for the single-pixel ImageNet attack.

The defense relies on two assumptions. First, the attacker can modify only the loss-computation code. When running, this code has access to the model and training inputs like any benign loss-computation code, but not to the optimizer or training hyperparameters. Second, the computational graph is trusted (e.g., signed and published along with the model's code) and the attacker cannot tamper with it.

We used Graphviz [23] to implement our prototype graph verification code. It lets the user visualize and compare computational graphs. The graph must be first built and checked by an expert, then serialized and signed. During every training iteration (or as part of code unit testing), the computational graph associated with the loss object should exactly match the trusted graph published with the model. The check must be performed for *every iteration* because backdoor attacks can be highly effective even if performed only in some iterations. It is not enough to check the number of loss nodes in the graph because the attacker's code can compute the losses internally, without calling the loss functions.

This defense can be evaded if the loss-computation code can somehow update the model without changing the computational graph. We are not aware of any way to do this efficiently while preserving the model's main-task accuracy.

## 8 Related Work

### 8.1 Backdoors

***Data poisoning.*** Based on poisoning attacks [2, 5, 6, 38], some backdoor attacks [10, 28, 48, 59] add mislabeled samples to the model's training data or apply backdoor patterns to the existing training inputs [47]. Another variant adds correctly labeled training inputs with backdoor patterns [70, 76, 92].

***Model poisoning and trojaning.*** Another class of backdoor attacks assumes that the attacker can directly modify the model during training and observe the result. Trojaning attacks [41, 55, 57, 58, 77] obtain the backdoor trigger by analyzing the model (similar to adversarial examples) or directly implant a malicious module into the model [86]; model-reuse

Table 6: **Comparison of backdoors and adversarial examples.**

| Features | Adversarial Examples | | Backdoors | | |
|---|---|---|---|---|---|
| | Non-universal [26, 64, 90] | Universal [8, 13, 46, 52, 61] | Poisoning [10, 28, 92] | Trojaning [29, 55, 103] | Blind (this paper) |
| Attacker's access to model | black-box [64], none* | black-box [52] | change data | change model | **change code** |
| Attack modifies model | no | no | yes | yes | **yes** |
| Inference-time access | required | required | required | required | **optional** |
| Universal and small pattern | no | no | yes | yes | **yes** |
| Complex behavior | limited [20] | no | no | no | **yes** |
| Known defenses | yes | yes | yes | yes | **no** |

* For an untargeted attack, which does not control the resulting label, it is possible to attack without model access [90].

attacks [40, 44, 98] train the model so that the backdoor survives transfer learning and fine-tuning. Lin et al. [49] demonstrated backdoor triggers composed of existing features, but the attacker must train the model and also modify the input scene at inference time.

Attacks of [51, 72, 101] assume that the attacker controls the hardware on which the model is trained and/or deployed. Recent work [14, 29] developed backdoored models that can switch between tasks under an exceptionally strong attack: the attacker's code must run concurrently with the model and modify the model's weights at inference time.

## 8.2 Adversarial examples

Adversarial examples in ML models have been a subject of much research [26, 43, 53, 65]. Table 6 summarizes the differences between different types of backdoor attacks and adversarial perturbations.

Although this connection is mostly unacknowledged in the backdoor literature, backdoors are closely related to UAPs, universal adversarial perturbations [61], and, specifically, adversarial patches [8]. UAPs require only white-box [8] or black-box [13, 52] access to the model. Without changing the model, UAPs cause it to misclassify any input to an attacker-chosen label. Pixel-pattern backdoors have the same effect but require the attacker to change the model, which is a strictly inferior threat model (see Section 2.5).

An important distinction from UAPs is that *backdoors need not require inference-time input modifications*. None of the prior work took advantage of this observation, and all previously proposed backdoors require the attacker to modify the digital or physical input to trigger the backdoor. The only exceptions are [3] (in the context of federated learning) and a concurrent work by Jagielski et al. [39], demonstrating a poisoning attack with inputs from a subpopulation where trigger features are already present.

Another advantage of backdoors is they can be much *smaller*. In Section 4.1, we showed how a blind attack can introduce a single-pixel backdoor into an ImageNet model. Backdoors can also trigger *complex functionality* in the model: see Sections 4.2 and 4.3. There exist adversarial examples that cause the model to perform a different task [20], but the perturbation covers almost 90% of the image.

In general, adversarial examples can be interpreted as features that the model treats as predictive of a certain class [35]. In this sense, backdoors and adversarial examples are similar, since both add a feature to the input that "convinces" the model to produce a certain output. Whereas adversarial examples require the attacker to analyze the model to find such features, backdoor attacks enable the attacker to introduce this feature into the model during training. Recent work showed that adversarial examples can help produce more effective backdoors [63], albeit in very simple models.

# 9 Conclusion

We demonstrated a new backdoor attack that compromises ML training code before the training data is available and before training starts. The attack is *blind*: the attacker does not need to observe the execution of his code, nor the weights of the backdoored model during or after training. The attack synthesizes poisoning inputs "on the fly," as the model is training, and uses multi-objective optimization to achieve high accuracy simultaneously on the main and backdoor tasks.

We showed how this attack can be used to inject single-pixel and physical backdoors into ImageNet models, backdoors that switch the model to a covert functionality, and backdoors that do not require the attacker to modify the input at inference time. We then demonstrated that code-poisoning attacks can evade any known defense, and proposed a new defense based on detecting deviations from the model's trusted computational graph.


# Acknowledgments

This research was supported in part by NSF grants 1704296 and 1916717, the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, and a Google Faculty Research Award. Thanks to Nicholas Carlini for shepherding this paper.

## References


[1] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. In *SysML*, 2019.

[2] Scott Alfeld, Xiaojin Zhu, and Paul Barford. Data poisoning attacks against autoregressive models. In *AAAI*, 2016.

[3] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *AISTATS*, 2020.

[4] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[5] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[6] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[7] Alex Birsan. Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies. The story of a novel supply chain attack. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, 2021.

[8] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. In *NIPS Workshops*, 2017.

[9] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. In *SafeAI@AAAI*, 2019.

[10] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv:1712.05526*, 2017.

[11] Ping-yeh Chiang, Renkun Ni, Ahmed Abdelkader, Chen Zhu, Christoph Studor, and Tom Goldstein. Certified defenses for adversarial patches. In *ICLR*, 2020.

[12] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. SentiNet: Detecting physical attacks against deep learning systems. In *DLS*, 2020.

[13] Kenneth T Co, Luis Muñoz-González, Sixte de Maupeou, and Emil C Lupu. Procedural noise adversarial examples for black-box attacks on deep convolutional networks. In *CCS*, 2019.

[14] Robby Costales, Chengzhi Mao, Raphael Norwitz, Bryan Kim, and Junfeng Yang. Live Trojan attacks on deep neural networks. In *CVPR Workshops*, 2020.

[15] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples. In *CVPR*, 2019.

[16] Jean-Antoine Désidéri. Multiple-gradient descent algorithm (MGDA) for multiobjective optimization. *Comptes Rendus Mathématique*, 350(5-6):313–318, 2012.

[17] Bao Gia Doan, Ehsan Abbasnejad, and Damith Ranasinghe. DeepCleanse: A black-box input sanitization framework against backdoor attacks on deep neural networks. *arXiv:1908.03369*, 2019.

[18] Min Du, Ruoxi Jia, and Dawn Song. Robust anomaly detection and backdoor attack detection via differential privacy. In *ICLR*, 2020.

[19] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *NDSS*, 2021.

[20] Gamaleldin F. Elsayed, Ian J. Goodfellow, and Jascha Sohl-Dickstein. Adversarial reprogramming of neural networks. In *ICLR*, 2019.

[21] N Benjamin Erichson, Dane Taylor, Qixuan Wu, and Michael W Mahoney. Noise-response analysis for rapid detection of backdoors in deep neural networks. *arXiv:2008.00123*, 2020.

[22] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Physical adversarial examples for object detectors. In *WOOT*, 2018.

[23] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. https://www.graphviz.org/pdf/dotguide.pdf, 2015.

[24] Yansong Gao, Chang Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. STRIP: A defence against trojan attacks on deep neural networks. In *ACSAC*, 2019.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[26] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[27] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. In *NIPS Workshops*, 2018.

[28] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying vulnerabilities in the machine learning model supply chain. In *NIPS Workshops*, 2017.

[29] Chuan Guo, Ruihan Wu, and Kilian Q Weinberger. TrojanNet: Embedding hidden Trojan horse models in neural networks. *arXiv:2002.10078*, 2020.

[30] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. TABOR: A highly accurate approach to inspecting and restoring trojan backdoors in AI systems. *arXiv:1908.01763*, 2019.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[32] Sanghyun Hong, Varun Chandrasekaran, Yiğitcan Kaya, Tudor Dumitraş, and Nicolas Papernot. On the effectiveness of mitigating data poisoning attacks with gradient shaping. *arXiv:2002.11497*, 2020.

[33] Jeremy Howard and Sylvain Gugger. fastai: A layered API for deep learning. *Information*, 11(2):108, 2020.

[34] Xijie Huang, Moustafa Alzantot, and Mani Srivastava. NeuronInspect: Detecting backdoors in neural networks via output explanations. *arXiv:1911.07399*, 2019.

[35] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. In *NeurIPS*, 2019.

[36] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[37] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *ICML*, 2013.

[38] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *S&P*, 2018.

[39] Matthew Jagielski, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. Subpopulation data poisoning attacks. *arXiv:2006.14026*, 2020.

[40] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-reuse attacks on deep learning systems. In *CCS*, 2018.

[41] Faiq Khalid, Muhammad Abdullah Hanif, Semeen Rehman, Rehan Ahmed, and Muhammad Shafique. TrISec: Training data-unaware imperceptible security attacks on deep neural networks. In *IOLTS*, 2019.

[42] Sergey Kolesnikov. Catalyst: Accelerated DL R&D. https://github.com/catalyst-team/catalyst, 2018.

[43] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *ICLR Workshops*, 2017.

[44] Keita Kurita, Paul Michel, and Graham Neubig. Weight poisoning attacks on pre-trained models. In *ACL*, 2020.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[46] Mark Lee and Zico Kolter. On physical adversarial patches for object detection. In *ICML Workshops*, 2019.

[47] Yiming Li, Tongqing Zhai, Baoyuan Wu, Yong Jiang, Zhifeng Li, and Shutao Xia. Rethinking the trigger of backdoor attack. *arXiv:2004.04692*, 2020.

[48] Cong Liao, Haoti Zhong, Anna Squicciarini, Sencun Zhu, and David Miller. Backdoor embedding in convolutional neural network models via invisible perturbation. In *CODASPY*, 2020.

[49] Junyu Lin, Lei Xu, Yingqi Liu, and Xiangyu Zhang. Composite backdoor attack for deep neural network by mixing existing benign features. In *CCS*, 2020.

[50] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, 2018.

[51] Tao Liu, Wujie Wen, and Yier Jen. SIN$^2$: Stealth infection on neural network - a low-cost agile neural Trojan attack methodology. In *HOST*, 2018.

[52] Xin Liu, Huanrui Yang, Ziwei Liu, Linghao Song, Hai Li, and Yiran Chen. DPatch: An adversarial patch attack on object detectors. In *AAAI Workshops*, 2018.

[53] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *ICLR*, 2017.

[54] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. ABS: Scanning neural networks for back-doors by artificial brain stimulation. In *CCS*, 2019.

[55] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2017.

[56] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv:1907.11692*, 2019.

[57] Yuntao Liu, Ankit Mondal, Abhishek Chakraborty, Michael Zuzak, Nina Jacobsen, Daniel Xing, and Ankur Srivastava. A survey on neural Trojans. In *ISQED*, 2020.

[58] Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural Trojans. In *ICCD*, 2017.

[59] Yuzhe Ma, Xiaojin Zhu, and Justin Hsu. Data poisoning against differentially-private learners: Attacks and defenses. In *IJCAI*, 2019.

[60] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *ACL*, 2011.

[61] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *CVPR*, 2017.

[62] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *NAACL-HLT: Demonstrations*, 2019.

[63] Ren Pang, Xinyang Zhang, Shouling Ji, Yevgeniy Vorobeychik, Xiaopu Luo, and Ting Wang. The tale of evil twins: Adversarial inputs versus backdoored models. In *CCS*, 2020.

[64] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *ASIACCS*, 2017.

[65] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, 2016.

[66] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Workshops*, 2017.

[67] Tomislav Peričin. SunBurst: the next level of stealth. https://blog.reversinglabs.com/blog/sunburst-the-next-level-of-stealth, 2019.

[68] PyTorch examples. https://github.com/pytorch/examples/, 2019.

[69] Ximing Qiao, Yukun Yang, and Hai Li. Defending neural backdoors via generative distribution modeling. In *NeurIPS*, 2019.

[70] Erwin Quiring and Konrad Rieck. Backdooring and poisoning neural networks with image-scaling attacks. In *DLS*, 2020.

[71] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *ICLR*, 2018.

[72] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. TBT: Targeted neural network attack with bit Trojan. *arXiv:1909.05193*, 2019.

[73] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv:1706.05098*, 2017.

[74] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[75] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *IJCV*, 115(3):211–252, 2015.

[76] Aniruddha Saha, Akshayvarun Subramanya, and Hamed Pirsiavash. Hidden trigger backdoor attacks. In *AAAI*, 2020.

[77] Ahmed Salem, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. Dynamic backdoor attacks against machine learning models. *arXiv:2003.03675*, 2020.

[78] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *NIPS*, 2018.

[79] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *CVPR*, 2015.

[80] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *ICCV*, 2017.

[81] Ozan Sener and Vladlen Koltun. Multi-task learning as multi-objective optimization. In *NIPS*, 2018.

[82] Ezekiel Soremekun, Sakshi Udeshi, Sudipta Chattopadhyay, and Andreas Zeller. Exposing backdoors in robust machine learning models. *arXiv:2003.00865*, 2020.

[83] Jacob Steinhardt, Pang Wei Koh, and Percy S Liang. Certified defenses for data poisoning attacks. In *NIPS*, 2017.

[84] Te Juin Lester Tan and Reza Shokri. Bypassing backdoor detection algorithms in deep learning. *arXiv:1905.13409*, 2019.

[85] Di Tang, XiaoFeng Wang, Haixu Tang, and Kehuan Zhang. Demon in the variant: Statistical analysis of DNNs for robust backdoor contamination detection. In *USENIX Security*, 2021.

[86] Ruixiang Tang, Mengnan Du, Ninghao Liu, Fan Yang, and Xia Hu. An embarrassingly simple approach for Trojan attack in deep neural networks. In *KDD*, 2020.

[87] Guanhong Tao, Shiqing Ma, Yingqi Liu, and Xiangyu Zhang. Attacks meet interpretability: Attribute-steered detection of adversarial samples. In *NIPS*, 2018.

[88] Florian Tramèr, Jens Behrmann, Nicholas Carlini, Nicolas Papernot, and Jörn-Henrik Jacobsen. Fundamental tradeoffs between invariance and sensitivity to adversarial perturbations. In *ICML*, 2020.

[89] Florian Tramèr and Dan Boneh. Adversarial training and robustness for multiple perturbations. In *NeurIPS*, 2019.

[90] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv:1704.03453*, 2017.

[91] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *NIPS*, 2018.

[92] Alexander Turner, Dimitris Tsipras, and Aleksander Madry. Clean-label backdoor attacks. https://openreview.net/forum?id=HJg6e2CcK7, 2018.

[93] Sakshi Udeshi, Shanshan Peng, Gerald Woo, Lionell Loh, Louth Rawshan, and Sudipta Chattopadhyay. Model agnostic defence against backdoor attacks in machine learning. *arXiv:1908.02203*, 2019.

[94] Akshaj Kumar Veldanda, Kang Liu, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. NNoculation: Broad spectrum and targeted treatment of backdoored DNNs. *arXiv:2002.08313*, 2020.

[95] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural Cleanse: Identifying and mitigating backdoor attacks in neural networks. In *S&P*, 2019.

[96] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. HuggingFace's transformers: State-of-the-art natural language processing. *arXiv:1910.03771*, 2019.

[97] Xiaojun Xu, Qi Wang, Huichen Li, Nikita Borisov, Carl A Gunter, and Bo Li. Detecting AI trojans using meta neural analysis. *arXiv:1910.03137*, 2019.

[98] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y Zhao. Regula sub-rosa: Latent backdoor attacks on deep neural networks. In *CCS*, 2019.

[99] Huan Zhang, Hongge Chen, Chaowei Xiao, Bo Li, Duane Boning, and Cho-Jui Hsieh. Towards stable and efficient training of verifiably robust neural networks. In *ICLR*, 2020.

[100] Ning Zhang, Manohar Paluri, Yaniv Taigman, Rob Fergus, and Lubomir Bourdev. Beyond frontal faces: Improving person recognition using multiple cues. In *CVPR*, 2015.

[101] Yang Zhao, Xing Hu, Shuangchen Li, Jing Ye, Lei Deng, Yu Ji, Jianyu Xu, Dong Wu, and Yuan Xie. Memory Trojan attack on neural network accelerators. In *DATE*, 2019.

[102] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and analyzing deep neural network training. In *IISWC*, 2018.

[103] Minhui Zou, Yang Shi, Chengliang Wang, Fangyu Li, WenZhan Song, and Yu Wang. PoTrojan: Powerful neural-level trojan designs in deep learning models. *arXiv:1802.03043*, 2018.

## A Example of a Malicious Loss Computation

Algorithm 2 shows an example attack compromising the loss-value computation of the RoBERTA model in HuggingFace Transformers repository. Transformers repo uses a separate class for each of its many models and computes the loss as part of the model's `forward` method. We include the code commit[5] that introduces the backdoor and passes all unit tests from the transformers repo.

The code computes the gradients and losses for every task and uses MGDA to obtain the scaling coefficients and compute the blind loss $\ell_{blind}$. The `forward` method then returns this loss value to the unmodified training code, which performs backpropagation and updates the model using the unmodified optimizer.

[5] https://git.io/Jt2fS.

**Algorithm 2** Blind attack on loss computation.

**Inputs:** model θ, dataset $\mathcal{D}$, optimizer *optim*.

▷ ***attacker-controlled code:***

**Auxiliary functions:** input synthesizer $\mu()$, label synthesizer $\nu()$, determine threshold *check_threshold*(), multiple gradient descent algorithm *MGDA*(), backpropagation function *get_grads*(), and loss *criterion*.

*# methods in the **RobertaForSequenceClassification** class*
**function** FORWARD(`self`, $x, y$)
  **if check_threshold**(`self.loss_hist`) **then**
    ***# no attack***
    $out$ = `self.roberta`($x$)  ▷ forward pass
    $logits$ = `self.classifier`($out$)
    $\ell$ = **criterion**($logits$, $y$)
  **else**
    ***# blind attack***
    $\ell_m, g_m$ = **`self`.get_loss_grads**($x, y$)
    $x^* = \mu(x)$
    $y^* = \nu(x, y)$
    $\ell_{m^*}, g_{m^*}$ = `self`.**get_loss_grads**($x^*, y^*$)
    $\alpha_0, \alpha_1$ = **MGDA**($[\ell_m, \ell_{m^*}], [g_m, g_{m^*}]$)
    $\ell_{blind} = \alpha_0 \ell_m + \alpha_1 \ell_{m^*}$
    $\ell = \ell_{blind}$
  `self.loss_hist.append`($\ell_m$)  ▷ save loss
  **return** $\ell$

**function** GET_LOSS_GRADS(`self`, $x, y$)
  $out$ = `self.roberta`($x$)
  $logits$ = `self.classifier`($out$)  ▷ forward pass
  $\ell$ = **criterion**($logits$, $y$)
  g = get_grads($\ell$, `self`)  ▷ backward pass
  **return** $\ell$, $g$

▷ ***Unmodified code:***

**function** TRAINER(RoBERTa model θ, dataset $\mathcal{D}$)
  **for** $x, y \leftarrow \mathcal{D}$ **do**
    $\ell$ = θ.**forward**($x, y$)
    $\ell$.backward()  ▷ backward pass
    optim.step()  ▷ update model
    θ.zero_grad()  ▷ clean model